Planned Programming Problem Gotchas as Lessons in Requirements Engineering

Daniel M. Berry, Craig S. Kaplan
Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada N2L 3G1
dberry@uwaterloo.ca, csk@uwaterloo.ca

***Abstract*—The term "gotcha" is used describe an unforeseen exception or complexity in a programming assignment given to students. All too often, the students' discovering gotchas in a programming assignment embarrasses the instructor into revising the assignment. This paper argues that students' discovery of gotchas should be encouraged to promote experiential learning of the value and necessity of requirements engineering. Rather than viewing the discovery experience as a misfortune, an instructor should welcome the experience and even plan assignments with an abundance of gotchas to be discovered by students.**

***Keywords*-classroom misfortunes, error checking, in-class discussion, introductory programming courses, learning experiences, mistakes in programming assignments, planning error checking, programming assignments, requirements engineering**

I. Background

Nearly every instructor who has given students a seemingly simple programming assignment has faced the situation of the students' discovering exceptions or complexities not even considered, let alone described, in the problem description. The exception or complexity identified in each of these discoveries is called a *gotcha*[1]. A gotcha is generally an unexpected *edge case* or *corner case* [1], [2].

On the reporting of each gotcha by a gleeful student, the typical instructor is embarrassed and apologetic for not having gotten the assignment right, updates the assignment, and gives the students more time to make up for the fact that more code has to be written. Even worse, part of the code already written by many a student has to be thrown out as useless. The instructor hopes and sometimes even promises that no more gotchas are on the horizon. If and when another gotcha is found, the instructor is even more embarrassed. The instructor privately vows to be more careful. After several gotchas, an occasional instructor even gives up and tells the students to write the program to the incorrect specification, promising to grade leniently. The students, and even the instructor, wonder why the instructor cannot get a simple programming assignment right.

[1]"Gotcha" comes from "got ya", which comes from "got you" and is used in sentences like "We gotcha!".

Indeed, this situation happened to the author Kaplan as he was teaching an introductory programming class titled "Introduction to Computer Science 1" (CS 115), taken generally by first year CS students at the University of Waterloo (UW) [3].

He had prepared what he thought was a very simple programming assignment: to construct a UW user's login name from his or her given and family names. At UW, a login name is usually chosen by concatenating all the initials of a user's given names together with his or her family name, and truncating the result to eight characters. Collisions among resulting eight-character sequences are resolved before truncation by inserting a serial number between the initials and the family name. He had planned for this programming problem to be given over a sequence of three assignments, each handling a larger set of cases. The first assignment took into account the two known gotchas, collisions and having only one name, by explicitly assuming them away. He intended to eliminate one gotcha in each subsequent assignment. However, as students were working on their programs, Kaplan and the students began to notice other, unplanned gotchas. These gotchas turned simple programming assignments that could be fulfilled by relatively exception-free programs into surprisingly complex assignments requiring programs that handled many exceptions to the normal case and to the exceptions. The assignments had become a hard-knocks lesson into the realities of software development.

The idea that artifacts should have gotchas is not new to teaching. Feather *et al.* suggest that exemplars for requirements engineering (RE) research should have "snares for the unwary" [4]. An exemplar, a prepublished standard problem to which an RE method can be applied for validation, that does not have gotchas is not doing its job of exercising the methods to be validated with it.

See the Appendix for cleaned up versions of the texts of the programming assignments and a list of the gotchas that the second author, his assistants, and his students found.

II. Proposal for Teaching

This paper argues that the instructor, instead of being embarrassed, should treat each discovery of a gotcha as a normal, natural part of software development and an opportunity to introduce issues of RE in the undergraduate classroom. Moreover, it even suggests that programming assignments in at least one early programming class in any software engineering (SE) program should be designed so that:

1) There is a sequence of programming assignments, each after the first being an enhancement of the previous, adding both new features and correcting gotchas discovered in the previous assignment.
2) The first assignment is to write software to solve a seemingly simple problem booby trapped with gotchas, ready to be discovered by the students while programming a solution to the problem.
3) While the sequence of enhancements may be planned ahead of time, the allocation of gotchas to assignments is done as they are discovered.
4) Any gotcha known by the instructor up front is not revealed to the students until the very last assignment if no student has found it by then.

III. Lessons Learned from the Sequence of Assignments

Such a sequence of programming assignments and all the gotchas that arise serve to teach the students a number of very important SE and RE lessons.

- No problem in real life is simple; there are always overlooked details that can mess up any solution. Some know this observation as Murphy's Law, "Anything that can go wrong will go wrong." [5] This lesson is learned directly from the relentlessness of the discoveries of gotchas.
- Each universal statement about a real-life phenomenon is only mostly true and sometimes false, and is therefore logically false, including this one[2] [6]! This lesson is learned from the discovered falseness of the implicit universal assumption inherent in the simple programming assignment, e.g., that for each user, the construction of the user's UW login name follows the simple rule given in the first assignment.
- Most of any software that solves a real-world problem is code to deal with exceptions. The exception handling code can be as much as 95% of the code of the software [7], [8]. This lesson is learned from the growing portion of each student's program that is dealing with the gotchas discovered so far.
- Tony Hoare's Law of Large Programs says, "Inside every large program is a small program struggling to get out." [9] This law says that if you have written a large, hairy, messy program to solve a problem, you need to step back, think about the problem in other ways until you find a view of the problem, with perhaps a different representation of the data, that allows a very simple algorithm to solve the problem. The very simple algorithm implemented is the small program that was struggling to get out of your original large program.
  On the other hand, there is a seemingly opposite law:

  > In every seemingly simple real-world problem committed to software is a big, hairy, exception-riddled, complex problem hiding, ready to wreak havoc on the software when least expected.

  Notice that the contradiction is only superficial. Hoare's law talks about programs and the new law talks about problems. Indeed, in the same e-mail in which Hoare confirmed the text of Hoare's law, Hoare added, "the small program can be found inside the large one only by ignoring the exceptions." So, it is more accurate to say that around every nice, clean, small program that does the job for 90% of the cases is a large, hairy, messy program that does the job for the remaining 10% of the cases. However, the small program is about 1/10 of the size of the large program [7]. This lesson is learned from the shrinking portion of each student's program that is the original code that he or she wrote to deal with the simple case described in the first assignment.
- In real life, no client or user desiring a new software to solve a previously unsolved problem, knows everything about the problem or even everything that the software should do, i.e., all the requirements, up front. An occasional exception is not even knowable until it happens. Often the client or user has to see *some* solution in operation to know what he or she really wants; he or she says "I'll know it when I see it (IKIWISI)." [10] This lesson is learned from the relentlessness of the discoveries of gotchas and the increasing uncertainty that the latest gotcha is the last that will be discovered.
- In real life, even software that has been adequately solving a problem for years, will occasionally be presented with a situation of which no one connected with the software ever even dreamt; Mother Nature has the last laugh [11]. This lesson is learned by extrapolating the lesson from the previous item several years down the line when a situation that happened not to happen before happens because, e.g., UW gets a student from a part of the world whose naming conventions are radically different from those of the rest of the world.
- Software that is running to solve a real-world problem ends up changing the real world to the extent that its own requirements change [12]. This lesson is learned from changes that may not happen in the assignment that the students have been given, simply because their

[2]One apparent exception to this statement is "Each human being is mortal", which is universally true.

programs are not used in the real world. The instructor may need to present an example drawn from the literature about real-world experiences. Nevertheless, the students should be in a position to believe the story and to understand the difficulties in modifying the involved software to deal with the changes.

- There are very early and recent data that show that a large majority, from 65% to 85%, of all defects found in running software can be traced back to requirements analysis time [13], [14] and that 70% to 85% of total project costs are rework due to requirements errors and new requirements [14]. Therefore, many but not all of the gotchas could have been discovered before coding began had enough time been set aside to think through the problem and to try to discover all the gotchas that are lurking in the problem [15], [16]. This lesson is learned from the fact that all the gotchas are problems with the original requirements. It should be easy to convince the students that with a little bit of thinking and maybe brainstorming before writing any code, they could have found many, if not all, of the gotchas that were discovered.
- The earlier a gotcha is found, the cheaper it is to fix. The cost to repair a defect grows exponentially with the lifecycle stage in which it is found. The later a defect is found, the more artifacts have been created that have components affected by the defect and that, therefore, have to be changed to fix the defect completely. Moreover, finding each affected component of each of these affected artifacts takes time. A defect caught after deployment of software can cost about 1.3 orders of magnitude more to fix than if it had been caught and fixed during requirements analysis [13], [17], [18]. This lesson is learned by extrapolation from data points obtained from the students when they are asked the question, "How long did it take you to revise your program after the discovery of each gotcha that caused a change to your program?"

These lessons are hard to learn. Witness the number of abandoned software projects, defective programs, and software disasters that stem from a failure to understand one of these lessons [19], [20]. These lessons are impossible to really teach in lectures or reading assignments; they must be learned from experience in order to be really believed to the point that they affect future behavior. It is up to programming instructors to give to students the experiences that teach them these lessons. Given the constraints of teaching a class with its time limitations, the experiences have to be small scale. Even a small scale experience will teach if the experience is real, is engaging, and produces enough frustration. The authors believe a planned sequence of assignments focused on one small problem from real life that has enough gotchas is sufficiently real. If the students are encouraged to find and report their own gotchas, the sequence will be sufficiently engaging. If deadlines have to slip to accommodate unplanned coding and if students have to discard code that they have already written, the sequence will engender enough frustration. Therefore, the authors believe that the sequence will teach the lessons well.

IV. Discussing the Lessons

It will be useful for each instructor to schedule a class meeting in which these lessons are *discussed* by the instructor and the students. That is, it is important that the students themselves describe the lessons that they have learned and that the instructor not try to lecture to them about the lessons. This meeting should start with the instructor asking the students how they felt, what frustrations they experienced, and what they learned during the sequence of assignments, especially when the gotchas necessitated rewriting program descriptions or code in mid-assignment. At some point in the discussion, the instructor should ask the students:

> If you had this much trouble with gotchas for such a small problem, can you imagine how many gotchas there are lurking in a large, multi-person–year industrial-sized problem?
>
> Can you imaging how expensive and frustrating dealing with these gotchas is in a large, multi-person–year industrial-sized problem?
>
> What percentage of failed software developments do you think failed because of unforeseen gotchas that proved too difficult to deal with when they were discovered?

The instructor should come prepared with a list of lessons to be shown in the last 10 minutes of the meeting to summarize the discussion and to at least describe any lesson that did not come up in the discussion.

V. Teaching the Lessons Early

The authors believe that these lessons should be taught as early as possible, even in the first programming courses. Just as the full requirements of a system should be determined early it its lifecycle, the student should learn the importance of early requirements determination early in his or her academic career.

Of course, the authors recognize that undergraduate courses, particularly at the first-year level, are already overloaded. The utility of introducing lessons from RE is clear, but the allocation of time to this material will be subject to constraints of time and curriculum.

Even if the entire recommendation cannot be followed, the authors encourage instructors to at least not be embarrassed when a student discovers a gotcha, to level with the students about how difficult it is to get assignments just right, and

thus to turn the discovery into a lesson about the realities of specifying software systems.

VI. Web Site for Gotchas

To help instructors prepare such sequences of assignments, the authors have created and will maintain a Web site, http://se.uwaterloo.ca/~dberry/Gotchas, containing problem descriptions, each of which is chock full of gotchas. The initial contents of this Web site is the problem described in this paper. The Web site provides instructions for visitors to submit their own gotcha-infested programming problems to help populate the site.

Acknowledgments

Daniel Berry's work was supported in parts by a Canadian NSERC grant NSERC-RGPIN227055-00.

Craig Kaplan's work was supported in part by a Canadian NSERC grant NSERC-RGPIN288206-00.

References

[1] Wikipedia, "Edge case," Viewed on 16 December 2009, http://en.wikipedia.org/wiki/Edge_case.

[2] ——, "Corner case," Viewed on 16 December 2009, http://en.wikipedia.org/wiki/Corner_case.

[3] University of Waterloo, "CS 115 Course Description," Waterloo, ON, Canada, Viewed on 30 October 2009, http://www.cs.uwaterloo.ca/current/courses/course_descriptions/cDescr/C%S115.shtml.

[4] M. S. Feather, S. Fickas, A. Finkelstein, and A. van Lamsweerde, "Requirements and specification exemplars," *Automated Software Engineering*, vol. 4, pp. 419–438, 1997.

[5] A. Bloch, *Murphy's Law and Other Reasons Why Things Go WRONG ("R" in "WRONG" written upside down)*. London, UK: Methuen, 1977.

[6] D. M. Berry and E. Kamsties, "The dangerous 'all' in specifications," in *Proceedings of 10th International Workshop on Software Specification & Design, IWSSD-10*. IEEE Computer Society Press, November 2000, pp. 191–194.

[7] F. P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*, 2nd ed. Reading, MA: Addison-Wesley, 1995.

[8] D. Berry, "The essential similarity and differences between mathematical modeling and programming," *Science of Computer Programming*, 2010, http://dx.doi.org/10.1016/j.scico.2010.05.002.

[9] C. A. R. Hoare, "Personal communication via electronic mail," August 2009.

[10] B. Boehm, "Requirements that handle IKIWISI, COTS, and rapid change," *IEEE Computer*, vol. 33, no. 7, pp. 99–102, 2000.

[11] D. C. Gause and B. Lawrence, "User-driven design," *Software Testing & Quality Engineering*, vol. 1, no. 1, pp. 23–27, 1999.

[12] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[13] B. W. Boehm, *Software Engineering Economics*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1981.

[14] K. Jackson, "Tutorial on requirements management and modeling with UML2," Tel Aviv, Israel, 2003.

[15] requirements-engineering.org, "RE Day 2005 Website," Viewed on 30 October 2009, http://www.requirements-engineering.org/REday05/.

[16] K. E. Wiegers, "When telepathy won't do: Requirements engineering key practices," Viewed on 30 October 2009, http://www.processimpact.com/articles/telepathy.html.

[17] S. R. Schach, *Software Engineering*, 2nd ed. Boston, MA, USA: Aksen Associates & Irwin, 1992.

[18] D. Leffingwell, "Calculating the return on investment from more effective requirements management," *American Programmer*, vol. 10, no. 4, pp. 13–16, 1997.

[19] The Standish Group, "The chaos report," 1994, http://www.standishgroup.com/sample_research/index.php.

[20] ——, "Extreme chaos," 2001, http://www.standishgroup.com/sample_research/index.php.

Appendix

The Actual Assignments

The sequence of three assignments that the second author ended up giving the students in the CS 115 class is given below[3]. Some minor typographical problems and inconsistencies in the vocabulary were cleaned up.

1) **Assignment 1:** A UWDir user ID string, hereinafter "ID", for a student is constructed from the student's first, middle, and last names by taking the first letter of the first name, the first letter of the middle name, and all the letters of the last name, with the restriction that the entire string is no longer than eight letters. For example, John Horatio Malkovich would be given the ID jhmalkov. (For simplicity, we assume that every student has a first name, a middle name, and a last name, and that no other student has a conflicting ID). Create a Scheme function make-uwdir-id that consumes three strings representing a student's first, middle and last

[3]By way of explanation:

- There were other assignments given in between the first and second of these, so that their assignment numbers are not consecutive.
- UWDir was, until recently, the human–machine system at the University of Waterloo (UW) that, among other things, assigns to each person, i.e., student, faculty, or staff, in the UW community, a user identification or login name that is unique across the campus so that a person may move freely among units on campus and keep the same user identification.
- A person's UWDir ID is this unique user identification.
- The UW campus mail system is set up so that for any UWDir ID id, id@uwaterloo.ca is always a correct e-mail address for id's owner and that any mail sent to id@uwaterloo.ca ends up being forwarded to whatever mailbox the owner prefers.
- Each student in the class, of course, has his or her own UWDir ID.

names, and produces a string containing an ID, constructed in the manner above. You may assume that all names are given entirely in lower case.

2) **Assignment 4:** In Assignment 1, we investigated the problem of constructing an ID from a student's names. In that case, we assumed that every student had a first, a middle, and a last name. In practice, however, a student may have any non-zero number of names. The ID is then constructed by concatenating the first letter of each name except the last, followed by all the letters of the last name, up to a maximum of eight letters. Thus, Moses would receive the ID moses, and John Ronald Reuel Tolkien would get jrrtolki.
   Create a Scheme function make-uwdir-id that consumes a list of strings representing all of a student's names, and produces a string containing an ID, constructed in the manner above. You may assume that all names are given entirely in lower case.
3) **Assignment 5:** In Assignments 1 and 4, we investigated the problem of constructing an ID from a student's names. We allowed any non-zero number of names, and constructed the ID by concatenating the first letter of each name except the last, followed by all the letters of the last name, up to a maximum of eight letters.
   We now extend the ID generating algorithm to deal with a list of students' names. When a particular ID is produced for the first time, we use the algorithm described previously. If a second student's names would result in the same ID, the number 2 is inserted before the first letter of the last name. The next student to produce the same ID will have 3 instead of 2, and so on. For example, Franklin Delano Roosevelt will get the ID fdroosev, Fiona Danielle Roosevelt will get the ID fd2roose and Fergus David Roosevelt will get the ID fd3roose. Note that if another student exists, with, e.g., the name Frederick Drooseveltry, his would-be ID fdroosev conflicts with the one already given to Franklin Delano Roosevelt, although he has only one first name. In this case, f2droose would be the correct ID to produce. However, also f4droose would be an acceptable ID to produce.
   Create a Scheme function make-uwdir-id that consumes a list of lists of strings, where each list of strings represents all of a student's names, and produces a list of strings containing IDs, constructed in the manner above. The order of the IDs should be the same as the order of student names in the original input list. You may assume that all names are given entirely in lower case, that a student never has more than 4 first names, and that there are no more than 20 students with identical IDs according to the initial rules.
   One approach to write this function is to do it in 3 phases. The first phase takes the list of lists of strings and builds a list of structures, one structure for every student. Each structure contains (1) a string, standing for a student's ID, built according to the initial rules of Assignment 4; (2) a number i, standing for the number of first names for this student; (3) and another number c that will count the number of students with the same initial ID seen so far. At the first phase, the c in each structure is 1. The second phase goes over the list of structures and updates the cs: the first occurrence of an ID will have c = 1, the second will have c = 2, and so on. Finally, the third phase translates the list of structures into a list of strings containing IDs, by inserting each c, other than 1, into the correct place in the corresponding initial ID. Note that each final ID must have at most 8 characters.

Commentary

A full list of the gotchas discovered by the second author, his assistants, and the students is:

- What if a student has only one name? Does it count as a first name or a last name? Perhaps Moses's ID should be m, considering moses to be his first name.
- What if a student has more than six middle names? In that case, truncation means that not all of the middle names are represented in the ID, and the last name might not appear at all.
- What if there are ten million students with the same names? Would the collision avoidance numbers completely overwhelm the IDs?
- What if there are one hundred million students with the same names? Then the IDs would be just numbers, and they would start conflicting with each other. Of course, in the limit, if you have more than 36^8 people, then you can't possibly keep them distinct with ISs consisting of 8 alphanumeric characters.
- Even if students don't have the same initials and last name, their names can still conflict. Let's say you have John Smith, James Smith, and Jane Susan Mith. The two Smiths would receive IDs jsmith and j2smith. Then, Jane has to be js2mith, even though there are no other Miths in the system.
- If multiple students have the same only one name, what do we do? Do we, for example, end up with moses and 2moses or with moses and moses2?

The reader can undoubtedly think of some more gotchas.

As can be seen, the more obvious gotchas, which were discovered by the instructor ahead of time, are already accounted for in the assignments, through a combination of explicit limitations on the input, e.g. "a student never has more than 4 first names" and leniency in the output, e.g.,"also f4droose would be an acceptable ID" Had there been time for more assignments in the term, some of the other gotchas could have been incorporated in later assignments.

Note that the idea of using gotchas pedagogically occurred to the authors only after the term was finished. Therefore, in the case of the reported experience, the gotchas were not planned into the assignments as suggested by the body of this paper. Were the gotchas properly planned, no assignment would mention any previously undiscovered gotcha. Rather, the students would be expected to find gotchas and report them. Whenever a gotcha is reported, the instructor would at that time decide, on the basis of time left until the current assignment's due date and pedagogical value, whether the gotcha ought to be handled in the current assignment or to be assumed away in the current assignment and included in a subsequent assignment.

In retrospect, the gotchas fall into two categories. One category is a fundamental problem arising at the extremes of the data, e.g., having more than 36^8 people. The other category is a more mundane, but nevertheless important, requirement problem, e.g, whether to produce 2moses or moses2 for the second occurrence of the one-named person Moses. While it may be easy to write the code for whatever choice is taken, the choice should follow some explicit requirements analysis done with the students' participation.